JSON

Version 8.16.0.2

Eli Barzilay
and Dave Herman

February 17, 2025

```
(require json)
```
package: base

This library provides utilities for parsing and producing data in the JSON data exchange format to/from Racket values. See the JSON web site and the JSON RFC for more information about JSON.

1 JS-Expressions

```
(jsexpr? x [#:null jsnull]) → boolean?
  x : any/c
  jsnull : any/c = (json-null)
```

Performs a deep check to determine whether *x* is a jsexpr.

This library defines a subset of Racket values that can be represented as JSON strings, and this predicates checks for such values. A *JS-Expression*, or *jsexpr*, is one of:

- the value of *jsnull*, `'null` by default, which is recognized using `eq?`
- `boolean?`
- `string?`
- `(or/c exact-integer? (and/c inexact-real? rational?))`
- `(listof jsexpr?)`
- `(hash/c symbol? jsexpr?)`

Examples:

```
> (jsexpr? 'null)
#t
> (jsexpr? #t)
#t
> (jsexpr? "cheesecake")
#t
> (jsexpr? 3.5)
#t
> (jsexpr? (list 18 'null #f))
#t
> (jsexpr? #hasheq((turnip . 82)))
#t
> (jsexpr? (vector 1 2 3 4))
#f
> (jsexpr? #hasheq(("turnip" . 82)))
#f
> (jsexpr? +inf.0)
#f
```

```
(json-null) → any/c
(json-null jsnull) → void?
  jsnull : any/c
```

This parameter determines the default Racket value that corresponds to a JSON "null". By default, it is the 'null symbol. In some cases a different value may better fit your needs, therefore all functions in this library accept a #:null keyword argument for the value that is used to represent a JSON "null", and this argument defaults to (json-null).

Note that the value of (json-null) (or an explicitly-provided #:null argument) is recognized using eq?.

2 Generating JSON Text from JS-Expressions

```
(write-json  x
             [out
              #:null jsnull
              #:encode encode
              #:indent indent]) → void?
  x : jsexpr?
  out : output-port? = (current-output-port)
  jsnull : any/c = (json-null)
  encode : (or/c 'control 'all) = 'control
  indent : (or/c #f #\tab natural-number/c) = #f
```

Writes the *x* jsexpr, encoded as JSON, to the *out* output port.

By default, only ASCII control characters are encoded as "\uHHHH". If *encode* is given as `'all`, then in addition to ASCII control characters, non-ASCII characters are encoded as well. This can be useful if you need to transport the text via channels that might not support UTF-8. Note that characters in the range of U+10000 and above are encoded as two \uHHHH escapes, see Section 2.5 of the JSON RFC.

If *indent* is provided and is not `#f`, each array element or object key–value pair is written on a new line, and the value of *indent* specifies the whitespace to be added for each level of nesting: either a `#\tab` character or, if *indent* is a number, the corresponding number of `#\space` characters.

Examples:

```
> (with-output-to-string
    (λ () (write-json #hasheq((waffle . (1 2 3))))))
"{\"waffle\":[1,2,3]}"
> (with-output-to-string
    (λ () (write-json #hasheq((와플 . (1 2 3)))
                      #:encode 'all)))
"{\"\\uc640\\ud50c\":[1,2,3]}"
> (for ([indent (in-list '(#f 0 4 #\tab))])
    (newline)
    (write-json #hasheq((waffle . (1 2 3)) (와플 . (1 2 3)))
                #:indent indent)
    (newline))

{"waffle":[1,2,3],"와플":[1,2,3]}

{
"waffle": [
1,
```

```
2,
3
],
"와플": [
1,
2,
3
]
}

{
    "waffle": [
        1,
        2,
        3
    ],
    "와플": [
        1,
        2,
        3
    ]
}

{
 "waffle": [
 1,
 2,
 3
 ],
 "와플": [
 1,
 2,
 3
 ]
}
```

```
(jsexpr->string  x
                [#:null jsnull
                 #:encode encode
                 #:indent indent]) → string?
  x : jsexpr?
  jsnull : any/c = (json-null)
  encode : (or/c 'control 'all) = 'control
  indent : (or/c #f #\tab natural-number/c) = #f
```

Generates a JSON source string for the jsexpr *x*.

Example:

```
> (jsexpr->string #hasheq((waffle . (1 2 3))))
"{\"waffle\":[1,2,3]}"
```

```
(jsexpr->bytes  x
                [#:null jsnull
                 #:encode encode
                 #:indent indent]) → bytes?
  x : jsexpr?
  jsnull : any/c = (json-null)
  encode : (or/c 'control 'all) = 'control
  indent : (or/c #f #\tab natural-number/c) = #f
```

Generates a JSON source byte string for the jsexpr *x*. (The byte string is encoded in UTF-8.)

Example:

```
> (jsexpr->bytes #hasheq((waffle . (1 2 3))))
#"{\"waffle\":[1,2,3]}"
```

3 Parsing JSON Text into JS-Expressions

```
(read-json
 [in
  #:null jsnull
  #:replace-malformed-surrogate? replace-malformed-surrogate?])
→ (or/c jsexpr? eof-object?)
 in : input-port? = (current-input-port)
 jsnull : any/c = (json-null)
 replace-malformed-surrogate? : any/c = #f
```

Reads a jsexpr from a single JSON-encoded input port *in* as a Racket (immutable) value, or produces `eof` if only whitespace remains. Like `read`, the function leaves all remaining characters in the port so that a second call can retrieve the remaining JSON input(s). If the JSON inputs aren't delimited per se (true, false, null), they must be separated by whitespace from the following JSON input. When *replace-malformed-surrogate?* is not `#f` an escaped malformed surrogate will be replaced with the unicode replacement character, otherwise `exn:fail:read` will be raised. Raises `exn:fail:read` if *in* is not at EOF and starts with malformed JSON (that is, no initial sequence of bytes in *in* can be parsed as JSON); see below for examples.

Examples:

```
> (with-input-from-string
    "{\"arr\" : [1, 2, 3, 4]}"
    (λ () (read-json)))
'#hasheq((arr . (1 2 3 4)))
> (with-input-from-string
    "\"sandwich\""
    (λ () (read-json)))
"sandwich"
> (with-input-from-string
    "true false"
    (λ () (list (read-json) (read-json))))
'(#t #f)
> (with-input-from-string
    "true[1,2,3]"
    (λ () (list (read-json) (read-json))))
'(#t (1 2 3))
> (with-input-from-string
    "true\"hello\""
    (λ () (list (read-json) (read-json))))
'(#t "hello")
> (with-input-from-string
    "\"world\"41"
    (λ () (list (read-json) (read-json))))
```

```
'("world" 41)
> (with-input-from-string
    "sandwich sandwich" ; invalid JSON
    (λ () (read-json)))
string::1: read-json: bad input starting #"sandwich
sandwich"
> (with-input-from-string
    "false sandwich" ; valid JSON prefix, invalid remainder is not
(immediately) problematic
    (λ () (read-json)))
#f
> (with-input-from-string
    "false42" ; invalid JSON text sequence
    (λ () (read-json)))
string::6: read-json: bad input starting #"false42"
> (with-input-from-string
    "false 42" ; valid JSON text sequence (notice the space)
    (λ () (list (read-json) (read-json))))
'(#f 42)
```

Changed in version 8.1.0.2 of package `base`: Adjusted the whitespace handling to reject whitespace that isn't either `#\space`, `#\tab`, `#\newline`, or `#\return`.

Changed in version 8.15.0.12 of package `base`: Added `#:replace-malformed-surrogate?`

```
(string->jsexpr str [#:null jsnull]) → jsexpr?
  str : string?
  jsnull : any/c = (json-null)
```

Parses a recognizable prefix of the string *str* as an immutable jsexpr. If the prefix isn't delimited per se (true, false, null), it must be separated by whitespace from the remaining characters. Raises `exn:fail:read` if the string is malformed JSON.

Example:

```
> (string->jsexpr "{\"pancake\" : 5, \"waffle\" : 7}")
'#hasheq((pancake . 5) (waffle . 7))
```

```
(bytes->jsexpr str [#:null jsnull]) → jsexpr?
  str : bytes?
  jsnull : any/c = (json-null)
```

Parses a recognizable prefix of the string *str* as an immutable jsexpr. If the prefix isn't delimited per se (true, false, null), it must be separated by whitespace from the remaining bytes. Raises `exn:fail:read` if the byte string is malformed JSON.

Example:

```
> (bytes->jsexpr #"{\"pancake\" : 5, \"waffle\" : 7}")
'#hasheq((pancake . 5) (waffle . 7))
```

4 Extension Procedures

```
(require (submod json for-extension))
```

The bindings documented in this section are provided by the `(submod json for-extension)` module, not `json`.

It may be more convenient for some programs to use a different representation of JSON than a `jsexpr?`. These procedures allow for customization of representation read or written. For example, in the Rhombus language it is more natural to use strings (which are immutable) for object keys, Rhombus lists (a.k.a. `treelist?`) for JSON lists, and immutable strings as JSON string values.

```
(write-json* who
             x
             out
             #:null jsnull
             #:encode encode
             #:indent indent
             #:object-rep? object-rep?
             #:object-rep->hash object-rep->hash
             #:list-rep? list-rep?
             #:list-rep->list list-rep->list
             #:key-rep? key-rep?
             #:key-rep->string key-rep->string
             #:string-rep? string-rep?
             #:string-rep->string string-rep->string) → void?
  who : symbol?
  x : any/c
  out : output-port?
  jsnull : any/c
  encode : (or/c 'control 'all)
  indent : (or/c #f #\tab natural-number/c)
  object-rep? : (-> any/c boolean?)
  object-rep->hash : (-> any/c hash?)
  list-rep? : (-> any/c boolean?)
  list-rep->list : (-> any/c list?)
  key-rep? : (-> any/c boolean?)
  key-rep->string : (-> any/c string?)
  string-rep? : (-> any/c boolean?)
  string-rep->string : (-> any/c string?)
```

Writes the value *x*, encoded as JSON, to the *out* output port.

The *who* argument is used for error reporting. The *jsnull*, *encode*, and *indent* arguments behave the same as described for `write-json`.

The *object-rep?* function should recognize values that will be written as JSON objects, and *object-rep->hash* must convert the value to a hash?.

The *list-rep?* function should recognize values that will be written as JSON arrays, and *list-rep->list* must convert the value to a list?.

The *key-rep?* function should recognize values that will be written as JSON object keys, and *key-rep->string* must convert the value to a string?.

The *string-rep?* function should recognize values that will be written as JSON strings, and *string-rep->string* must convert the value to a string?.

Added in version 8.15.0.12.

```
(read-json*
 who
 in
 [#:replace-malformed-surrogate? replace-malformed-surrogate?]
 #:null jsnull
 #:make-object make-object-rep
 #:make-list make-list-rep
 #:make-key make-key-rep
 #:make-string make-string-rep)
→ any/c
 who : symbol?
 in : input-port?
 replace-malformed-surrogate? : any/c = #f
 jsnull : any/c
 make-object-rep : (-> (listof pair?) any/c)
 make-list-rep : (-> list? any/c)
 make-key-rep : (-> string? any/c)
 make-string-rep : (-> string? any/c)
```

Reads a value from a single JSON-encoded input port *in* as a Racket value, or produces eof if only whitespace.

The *who* argument is used for error reporting. The *jsnull* and *replace-malformed-surrogate?* arguments behave the same as described in read-json.

The *make-object-rep* receives a list? of key-value pairs, and returns a custom representation of a JSON object.

The *make-list-rep* receives a list? of values and returns a custom representation of a JSON array.

The *make-key-rep* receives a string? values for an object key, and returns a custom representation of a JSON key.

The *make-string-rep* receives a string? value, and returns a custom representation of a JSON string.

Added in version 8.15.0.12.

5 A Word About Design

5.1 The JS-Expression Data Type

JSON syntactically distinguishes "`null`", array literals, and object literals, and therefore there is a question of what Racket value should represent a JSON "`null`". This library uses the Racket `'null` symbol by default. Note that this is unambiguous, since Racket symbols are used only as object keys, which are required to be strings in JSON.

Several other options have been used by various libraries. For example, Dave Herman's PLaneT library (which has been the basis for this library) uses the `#\nul` character, other libraries for Racket and other Lisps use `(void)`, `NIL` (some use it also for JSON "`false`"), and more. The approach taken by this library is to use a keyword argument for all functions, with a parameter that determines its default, making it easy to use any value that fits your needs.

The JSON RFC only states that object literal expressions "SHOULD" contain unique keys, but does not proscribe them entirely. Looking at existing practice, it appears that popular JSON libraries parse object literals with duplicate keys by simply picking one of the key-value pairs and discarding the others with the same key. This behavior is naturally paralleled by Racket hash tables, making them a natural analog.

Finally, the JSON RFC is almost completely silent about the order of key-value pairs. While the RFC only specifies the syntax of JSON, which of course always must represent object literals as an ordered collection, the introduction states:

> An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.

In practice, JSON libraries discard the order of object literals in parsed JSON text and make no guarantees about the order of generated object literals, usually using a hash table of some flavor as a natural choice. We therefore do so as well.

5.2 Naming Conventions

Some names in this library use "jsexpr" and some use "json". The rationale that the first is used for our representation, and the second is used as information that is received from or sent to the outside world.